# pArray as an efficient static parallel container in STAPL

Olga V. Tkachyshyn[2] Author
Ping An[1] Graduate Student Mentor
Gabriel Tanase[1] Graduate Student Mentor
Nancy M. Amato[1] Faculty Mentor

Technical Report TR03-003
PARASOL Lab., Department of Computer Science
Texas A&M University
{olgat,pinga,gabrielt,amato}@cs.tamu.edu

Summer 2003

[1]Department of Computer Science, Texas A&M University, College Station, TX 77843.
[2]Department of Computer Science, Western Oregon University, Monmouth, OR 97338.

**Abstract:** *Standard Template Adaptive Parallel Library (STAPL) is a parallel library for C++ that allows users to execute programs on multiprocessor systems without having to deal with the complexity of application parallelization. The simplest data structure STAPL currently provides is a parallel vector, a hard to optimize dynamic data structure. This paper will describe a pArray, a new parallel data structure in STAPL that is static and yet sufficient for many applications and provides the highest possible efficiency. pArray builds upon the valarray data structure provided by Standard Template Library (STL). The basic design will be discussed and performance comparisons with parallel vector will be provided.*

# 1 Motivation

Modern computer applications like computational physics, computational biology, etc., demand speedy processing of large amounts of data. Using more than one processor at a time greatly increases the speed of program execution, making parallel and distributed processing important.

In sequential computing, standardized libraries have proved to be valuable tools for simplifying the program development process, by providing routines for common operations that allow programmers to concentrate on higher level problems. Similarly, libraries of basic and generic parallel algorithms provide important building blocks for parallel applications.

STAPL (Standard Template Adaptive Parallel Library) was designed to liberate programmers from the complexity of the parallel programming [1, 3]. STAPL is a parallel C++ library with functionality similar to STL, the widely accepted C++ Standard Template Library. STAPL provides generic parallel and distributed data structures called pContainers. The simplest pContainer STAPL currently provides is a pVector (parallel vector). Since pVector is a dynamic data structure (elements can be inserted and deleted at runtime), it is not the most efficient pContainer if all the user needs is a static data structure (the size stays the same throughout execution). Static data structures like array are common to scientific applications. This paper will describe a pArray, a new pContainer in STAPL that provides the highest possible efficiency of a static parallel data structure. pArray is built upon valarray, the equivalent sequential container provided by STL.

STAPL Overview is discussed in Section 2. STAPL provides distributed data structures with parallel methods, called pContainers (parallel containers), described in Section 3. The basic design of a pArray will be discussed in Section 4, a parallel algorithm implemented described in Section 5, and performance comparisons with pVector as well as the speedup information will be provided in Section 6.

# 2 STAPL Overview

STL is a widely used C++ library. It consists of three major components: *containers, algorithms and iterators.* Containers are data structures such as vector, list, map and set. Algorithms are operations such as searching, sorting and merging. Algorithms can operate on a variety of different containers using iterators as the interface. Iterators are generalized C++ pointers that abstract the type of container they traverse.

| STL | STAPL |
|---|---|
| Container | pContainer |
| Iterator | pRange |
| Algorithms | pAlgorithms |

Table 1: Comparison of STL and STAPL components

STAPL is a parallel version of STL and it consists of the following major components: *pContainers, pAlgorithms, pRanges, aRMI, Executor and Scheduler.* pContainers and pAlgorithms are the counterparts of the STL containers and algorithms. The pRange is a construct that presents an abstract view of data space which allows random access to elements of the pContainer. The STAPL Executor is responsible for executing the subranges of the pRange in the execution order is based on the data dependence information. RMI, or Remote Method Invocation, is the communication standard in STAPL. A high level comparison between STL and STAPL is summarized in Table 1.

# 3 pContainer Design

A pContainer [2] is the parallel equivalent of the STL container. The data is stored in the pContainer in a distributed fashion (i.e it can reside on multiple machines). The user can specify the distribution of data or STAPL can dynamically choose one for the user. Each pContainer provides methods to access the encompassed data. The same methods are used to access the data that is located in the local and remote processors. Inside these methods the pContainer determines whether the data is located locally or remotely. If it is remote, the pContainer fetches the data from the remote processor using RMI. The task of finding out whether the data is located locally or not falls to the *pContainer Distribution.* The Distribution Manager assigns to each element of the pContainer a unique global identifier called the *GID.* The Distribution Manager maintains a distributed lookup table that stores information about the actual location of each data element in the pContainer. pContainer uses this table in the Distribution Manager to find the exact location of the requested data.

The following sections describe the three base classes that provide the implementation interfaces and the functionality common to all pContainers.

## 3.1 Base pContainer

The Base pContainer class provides generic methods to construct, add, access, modify, delete, and locate elements. Each element must have a unique global identifier (GID). Using its Distribution Manager, the Base pContainer methods IsLocal(GID) and Lookup(GID) are used to directly locate any element. IsLocal(GID) checks if the GID is in the local processor, while Lookup(GID) finds the *location* of the element with that GID if it's not local.

## 3.2 Base Distribution Manager

The Distribution Manager enables the pContainer to access elements based on their GIDs, regardless of whether the object is local or remote. The location information is distributed across the system. A centralized element locator is non-scalable, and can easily become a performance bottleneck. Instead, each processor is responsible for storing the location information of a set of elements in an *element location map*, contained in its Base Distribution class. The designated processor for an element with a particular GID is determined by hashing.

## 3.3 Base Sequential Container Part

A parallel and distributed container is composed from a set of sequential containers. An abstract Base Sequential Container class serves to assure that a minimum set of methods, e.g. get, set, add, and delete elements, is provided by each Sequential Container with unified interface.

# 4 pArray as an example of a pContainer

pArray is a specific pContainer in STAPL. It is derived from the Base pContainer class and specialized with the proper Distribution Manager (Array Distribution) and the corresponding Sequential Container class (Array Part). Figure 3 presents the major components used to build a pArray and their relationships with each other. pArray can use the index as an implicit GID without actually storing it.

## 4.1 Array Part

pArray object stores a collection of sequential parts, called Array Parts. ArrayPart is consistent with the requirements of the Base Sequential Container and serves as a wrapper over the sequential STL container valarray. This is achieved by deriving the ArrayPart from the BasePart and instantiating a valarray in the ArrayPart class. ArrayPart has all of the functionality of the valarray.

## 4.2 Array Distribution

Array Distribution is a specific Distribution Manager responsible for locating local and remote elements. Therefore the major methods of the Array Distribution are *bool IsLocal(GID, PARTID)* (returns true if the element is in the local part) and *Location Lookup(GID)* (finds the owner of the GID). Two methods to manage the element distribution are described in the next subsections.

### 4.2.1 Decentralized Distribution Information

For the Decentralized Distribution method, the $n$ elements of the pArray stored on *nprocs* processors are divided into segments of the size *n/nprocs*. Each processor is responsible for

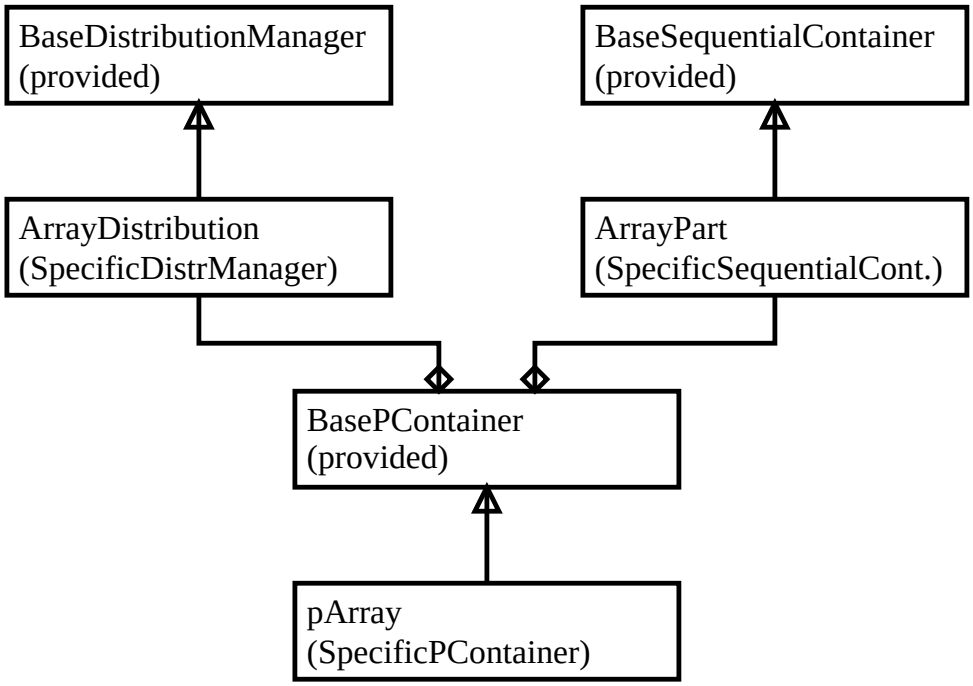


Figure 1: Class Hierarchy of pArray Components

knowing the location of elements in one segment. The lookup algorithm is the following:

```
T GetElement(GID id) {
  if(id is in local processor){
    call local GetElement;
    return the data;
  }
  else{
    if(id is in the location cache){
      lookup the map owner;
      request data from the data owner;
      return the data;
    }
    else{
      calculate the map owner = id*nprocs/n;
      request the location information from the map owner;
      cache the location information in the location cache;
      request data from the data owner;
      return data;
    }
  }
}
```

The Element Location System is demonstrated in Figure 2. For example, if the processor 0 needs an element with GID = 5, it will first check if the GID is in its own data. If the processor doesn't own the element, it will check the element location cache. If the location information is not there, the owner of the location map will be determined by $PID = GID * nprocs/n$ which in this case is $5 * 3/10 = 1$. Now processor 0 will request the location information from the processor 1 and cache it in its own location cache.

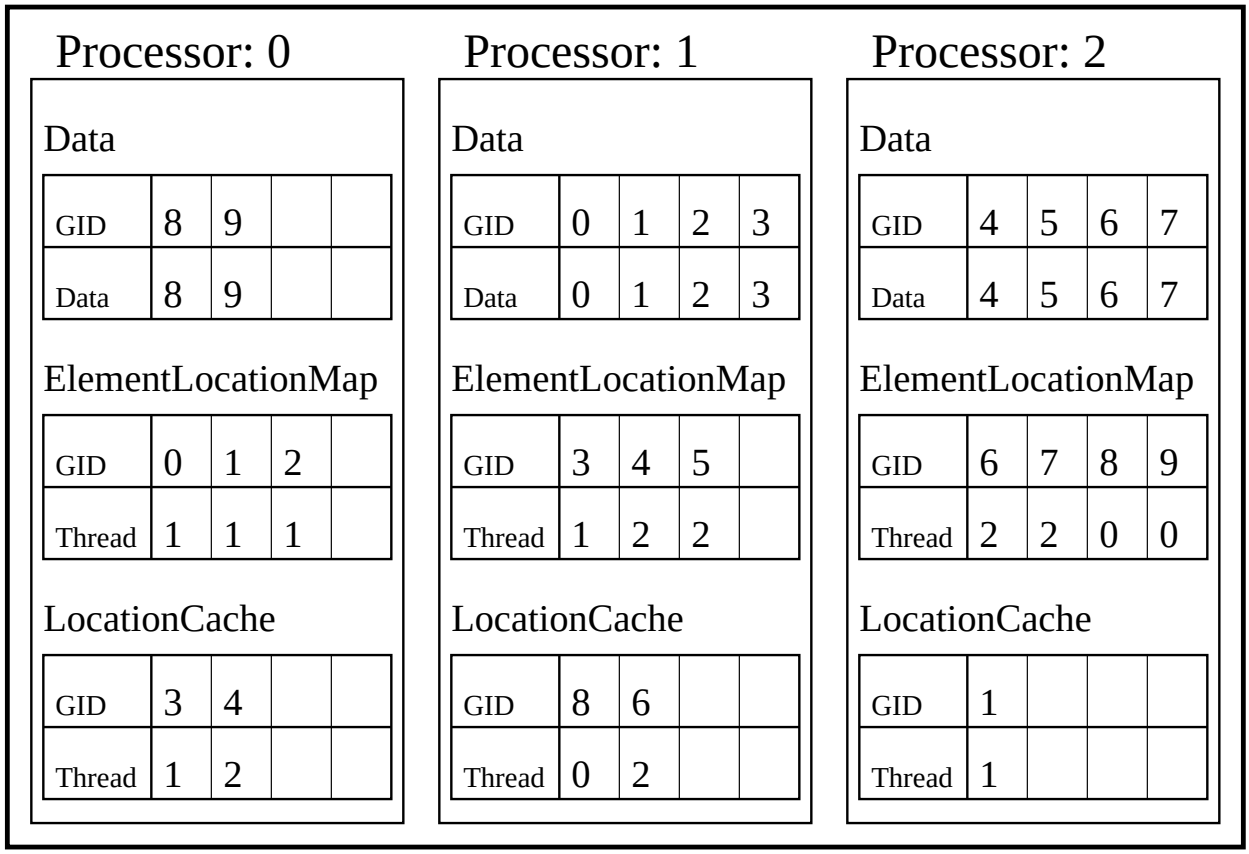


Figure 2: The Element Location System for Decentralized Distribution Information

This method avoids accessing bottleneck by distributing the location information between all processor and saves space by not duplicating any of the location information. But it can be slow since the location information may need to be requested remotely.

### 4.2.2 Duplicated Distribution Information

The size of pArray is fixed, allowing us to optimize the array distribution. In Duplicated Distribution each sequential part of data is described by a pair that consists of a starting index of the part and the size of the part. Array Distribution Vector stores these pairs together with the processor IDs of data owners. Each processor has a copy of the distribution vector and searches it for the range the specific GID is in when the processor needs to locate the element.

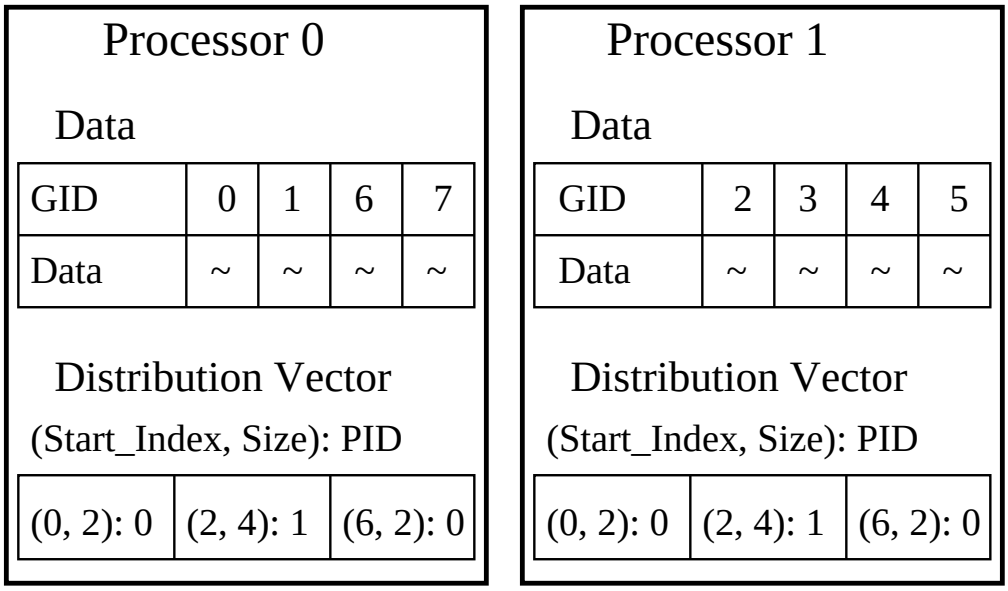


Figure 3: Class Hierarchy of pArray Components

For example, if the processor 0 needs an element with GID = 5, it will look in the Distribution Vector for the range that contains GID = 5, which in this case is the range that starts with GID = 2 and has the size 4. Once the range that includes the GID is determined, the data owner is looked up and the data is requested.

This method can be fast because all of the distribution information is local and there is no need to request location information remotely. On the other hand, the same distribution information is stored in multiple processors, which could be space consuming. Also, the distribution vector can be rather large and time consuming to search.

### 4.3 pArray

Base pContainer instantiates an ArrayPart (a specific sequential container derived from the Base SequentialContainer) and an ArrayDistribution (a specific Distribution Manager derived from BaseDistribution). pArray is then derived from the BasePContainer to implement the functionality specific to the pArray. pArray class provides the basic functionality of a parallel array, such as constructing an array, accessing and updating elements, locating remote elements. The pArray class is given below.

```
template< BasePContainer< ArrayPart<Data>, ArrayDistribution > >
class pArray
{
   //constructors
   pArray();                    //default constructor
   pArray(int size);            //default distribution
   pArray(int size, ArrayDistribution distr); //specified distribution
```

```
   //element access methods
   Data GetElement(GID);        //returns an element with specified GID
   void SetElement(GID,Data);   //sets specified location with given value

   //operators and array specific methods
   Data operator[];                    //index array access operator
   pArray operator+(Data scalar);      //adds a scalar to the pArray
   pArray operator+(pArray array);     //adds term by term two pArrays of the
                                       //same size (undefined otherwise);
                                       //returns an array with the same
                                       //distribution as the calling array
   pArray operator*(Data scalar);      //multiplies the pArray by a scalar
   pArray operator*(pArray array);     //multiplies term by term two pArrays of
                                       //the same size (undefined otherwise);
                                       //returns an array with the same
                                       //distribution as the calling array
   Data accumulate();                  //sums up all values stored in pArray
   Data dotproduct(pArray array);      //dot product of two pArrays of the
                                       //same size (undefined otherwise)
   long double euclideannorm();        //euclidean norm of an pArray
}
```

# 5 Prefix Sums as an example of a pAlgorithm

A pAlgorithm is the parallel counterpart of the STL algorithm. STAPL algorithms take pRanges as parameters. In the same way as iterators are used to access the elements of STL containers, pRanges allows the programmer to work with different containers in a uniform manner. We implemented a prefix sums algorithm that takes a pRange as an input and thus works for all the pContainers.

Prefix sums is one of the most basic parallel algorithms. It is a subroutine for various other parallel algorithms like sorts. Prefix sums of a sequence $S = x_1, x_2, ..., x_n$ of $n$ elements are the $n$ partial sums defined by $P_i = x_1 + x_2 + ... + x_i, 1 \leq i \leq n$.

The parallel prefix sums algorithm is as follows:

```
1.   all processors sum up their array parts in parallel
     using stl::accumulate()
2.   data exchange phase
2.1. each processor sends its part sum to processor 0
2.2. processor 0 calculates the starting sums for all processors
     using stl::partial_sum()
2.3. processor 0 sends appropriate starting sums to all processors
3.   all processors in parallel get prefix sums for their parts
     using stl::partial_sum()
```

# 6 Performance Results

We did preliminary experiments to test the performance of the pArray. We tested how the performance scales as the number of processors increases. Since pArray and pVector are very similar data structures, we compared their performance in prefix sums algorithm. More tests are needed to show the true advantages of the pArray.

## 6.1 Scalability

Scalability is the ability of a program to exhibit good speedup as the number of processors increases. $Scalability = Running\ Time\ On\ 1\ Processor / Parallel\ Running\ Time$. Scalability is one of the most common ways to measure the performance of parallel programming. Figure 4 demonstrates scalability of Prefix Sums using a pArray of 1,000,000 elements on 1 to 6 processors.

## 6.2 pArray vs. pVector

To compare the efficiency of the pArray and the pVector, we timed the prefix sums algorithm first using the pArray and then using the pVector of 1,000,000 elements. Since pArray is a static data structure and pVector is a dynamic data structure that provides methods to insert and delete elements, we expected pArray to be more efficient.

Figure 5 demonstrates the difference in the execution time; pArray is definitely faster than pVector due to less overhead.

# 7 Conclusion

STAPL enables users to write parallel programs easily and efficiently. STAPL eliminates the complexity of running programs on different platforms. pArray is one of the pContainers STAPL provides, it is a useful and efficient parallel container that shows good scalability. pArray is faster than pVector when data size is fixed due to less overhead. Parallel Prefix Sums is an efficient pAlgorithm.

Future work includes array redistribution, optimizing prefix sums for a large number of processors, as well as developing more pAlgorithms and more pContainers.

# 8 Biography

Olga Tkachyshyn is an undergraduate student at Western Oregon University. She is expecting to graduate in the Summer of 2004 with a Bachelor of Science in Computer Science and Mathematics degree, minor in Business. She is currently participating in the Distributed Mentor Project (DMP) at Texas A&M University.

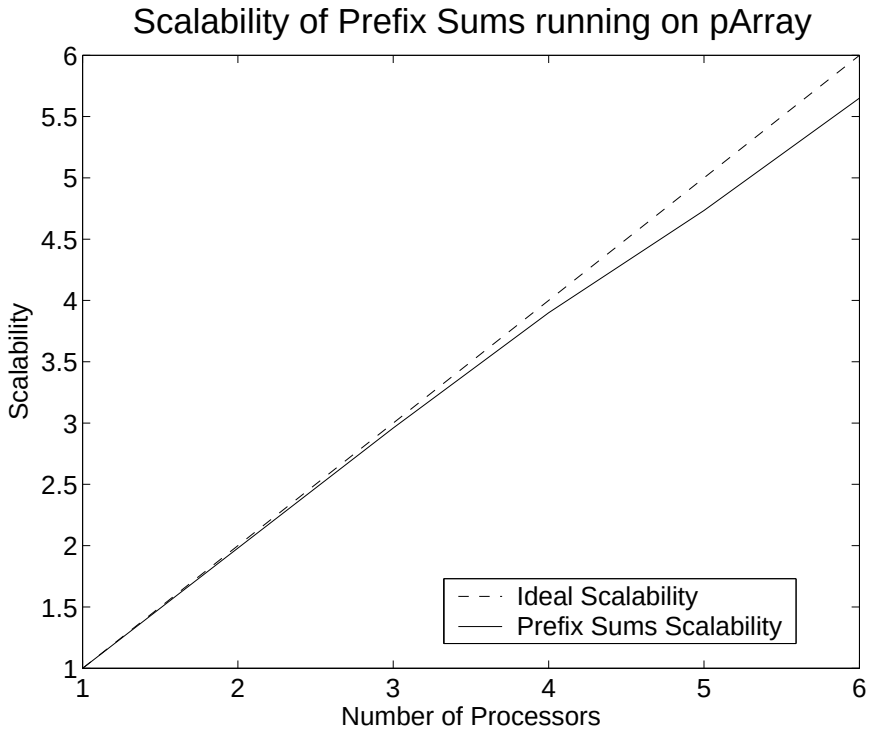


Figure 4: Scalability for Prefix Sums for a pArray of 1,000,000 elements

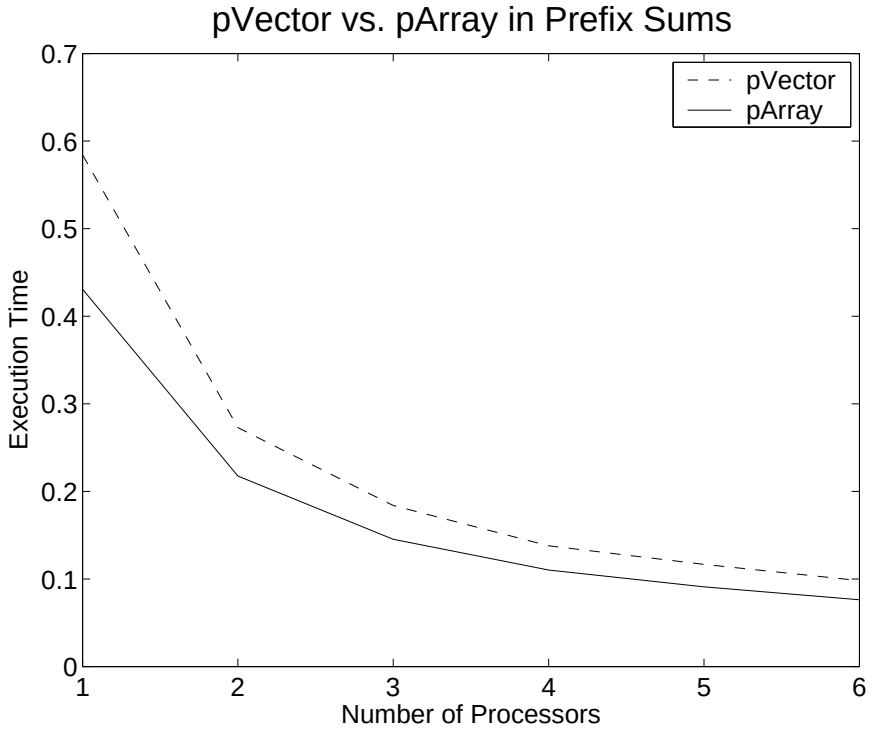


Figure 5: The Comparison of Program Running Time using pArray and pVector

# References


[1] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel programming library for C++. In *Proc. of the 14th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Cumberland Falls, Kentucky, Aug 2001.

[2] P. An, A. Jula, G. Tanase, P. Thomas, N. Amato, and L. Rauchwerger. *Efficient Parallel Containers with Shared Object View*, Aug 2003.

[3] L. Rauchwerger, F. Arzu, and K. Ouchi. Standard Templates Adaptive Parallel Library. In *Proc. of the 4th International Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers (LCR)*, Pittsburgh, PA, May 1998.